



**Università degli Studi di Bologna**  
**Facoltà di Ingegneria**

**Corso di  
Reti di Calcolatori T**

*Integrazione di Sistemi e  
Uso oculato delle risorse*

**Antonio Corradi, Luca Foschini**

**Giuseppe Martuscelli, Lorenzo Rosa, Marco Torello**

**Anno Accademico 2022/2023**

# **Alcuni concetti**

---

**Programmazione applicativa**

**Programmazione di sistema**

**Programmazione di sistema e consapevole delle risorse (resource-aware)**

**Programmi nel piccolo (in-the-small)**

**Programmi nel largo (in-the-large)**

**Programmazione imperativa**

**Programmazione dichiarativa**

# Programmazione di sistema

---

Accanto alla **programmazione applicativa**, fatta per raggiungere obiettivi specifici e costituita da applicazioni che rispondono a **esigenze utente** specifiche, i servizi applicativi non potrebbero funzionare senza un **supporto di programmi** che consentono di preparare l'ambiente e forniscono le **funzionalità di servizio**, che chiamiamo **componenti di sistema (non solo sistema operativo)**

In generale, i **componenti di sistema** tendono ad eseguire **molte volte, molto spesso, e devono essere ottimizzati**

In particolare, tendiamo ad avere **programmi piccoli** che si **devono coordinare per realizzare protocolli comuni e il migliore uso delle risorse**

# Consapevolezza delle Risorse

---

In generale, ogni programma che si debba mettere in esecuzione ha **bisogno di risorse di sistema di basso e alto livello** come **Processori, Processi, Memoria, I/O e Comunicazione, Dati, Codice, File...**

Anche nei sistemi concentrati (**unico processore**)

**Da PERSEGUIRE -- RESOURCE AWARENESS**

questo significa fare **programmi piccoli, efficienti e molto capaci di interagire in modo dinamico**

Uso delle **risorse della architettura in modo ottimale**, senza sprechi non necessari, con buon **controllo delle situazioni di errore**

**Da NON FARE**

**NON usare risorse complesse e pesanti e difficili da portare tra architetture e da implementare**

**NON fare algoritmi oscuri e troppo complessi**

# Modelli di architettura della soluzione

---

In generale, nella espressione dei programmi ci sono due paradigmi ispiratori:

**Programmazione Imperativa**

**Programmazione Dichiarativa**

**Modalità Imperativa o algoritmica**

La prima descrive i linguaggi tradizionali e anche ad oggetti con la **specifica di algoritmi precisi di soluzione**

**Modalità Dichiarativa o meno guidata dal progettista**

La seconda fa riferimento a modi diversi di **esprimere la computazione in modo meno deterministico**

# Modelli dichiarativi

---

## Programmazione dichiarativa

Questo paradigma porta ad esprimere soluzioni che devono soddisfare **una serie di requisiti specificati dall'utilizzatore**, **senza arrivare alla specifica completa di algoritmo**, che spesso può esplorare soluzioni molteplici e viene guidato **da non determinismo e da modalità ad elevato innestamento funzionale o logico garantito da un motore** alla base del supporto

I linguaggi sono basati **su diversi paradigmi**,  
vedi LISP, Prolog, Scala,...

La **specifiche esecuzione** viene lasciata alla libertà di supporto del **motore di base**, che potrebbe essere logico, funzionale, ecc.

**Noi non ci occuperemo di questa**

# Modelli imperativi

---

## Programmazione imperativa (codice e dati)

In questo paradigma, che ispira tutti i linguaggi che avete visto, **le istruzioni specificano completamente l'algoritmo** da risolvere senza lasciare ‘libertà’ di cercare soluzioni non specificate

L'**algoritmo** viene specificato come **sequenza (più o meno) precisa** di passi che vengono **eseguiti una dopo l'altra** e che agiscono su **dati, intesi come contenitori** in memoria delle informazioni

(in genere è il sistema operativo che inserisce poi la **possibilità di concorrenza, parallelismo, ...**)

Vedi C, C++, Java, C#, ...

Tutti questi **linguaggi usano stato** (inteso come **variabili - e loro tipi**) che viene **manipolato dagli algoritmi**

# Modelli imperativi

---

**La programmazione imperativa** prevede **codice e dati con la massima semplicità per programmazione di sistema**

**OBIETTIVO:** minimizzare il costo di esecuzione del programma, in algoritmo e risorse impegnate, e aumentare la solidità e la comprensibilità del programma

## C e Java

**Esempi:**

- **variabili statiche o dinamiche?**
- **procedure o programmi in linea? (specie se piccoli programmi e non invocati da punti diversi)**
- **leggibilità dei programmi (evitare break non necessari, aggiungere commenti necessari, ...)**

# Modelli per la granularità della soluzione

---

In generale, possiamo fare programmi ed applicazioni di grandi dimensioni o di dimensioni più limitate:

## **Programmi nel piccolo (in the small) (CASO di RETI)**

Applicazione di **dimensioni limitate** e con un **uso altrettanto limitato di risorse**

## **Programmi di grandi dimensioni (in the large)**

Qui intendiamo soluzioni **con grandi dimensioni** e con obiettivo di fornire soluzioni ad **ampie applicazioni e larghi requisiti di una organizzazione**

# **Architettura delle soluzioni**

---

**Nei sistemi distribuiti ancora di più si devono considerare le risorse messe in gioco ed impegnate**

**Sviluppo statico della applicazione**

**Supporto dinamico durante la esecuzione**

**Architettura logica della soluzione**

**Architettura del supporto alla applicazione**

**Modello di sviluppo integrato**

**Integrazione dei sistemi e dei componenti**

**Sistemi a livelli e consapevolezza reciproca**

# ARCHITETTURA logica e deployment

---

## Per una applicazione

- 1) *Design dell'algoritmo di soluzione*
- 2) *Codifica in uno o più linguaggi di programmazione*  
dobbiamo usare le specifiche dei linguaggi e definire le  
**risorse logiche del programma**

### 3) **Obiettivo: arrivare ad eseguire**

Per arrivare alla esecuzione, dobbiamo stabilire quali sono le  
**risorse hardware e fisiche** su cui potere eseguire

Dobbiamo attuare una corrispondenza **tra la parte logica e la  
parte concreta di architettura**

Questa architettura scelta con la sua **specifica configurazione** è  
la scelta di **deployment**

Il **deployment** è la fase in cui decidiamo come **allocare le risorse  
logiche alle risorse fisiche disponibili** (che abbiamo preparato  
decidendo **una configurazione di esecuzione**)

# ARCHITETTURA SUPPORTO

---

## Fasi Statiche (prima della esecuzione)

**Da una applicazione partiamo dall'algoritmo**

Analisi, sviluppo, e progetto dell'algoritmo e sua codifica in uno o più linguaggi di programmazione

Una volta prodotta ***una forma eseguibile***, noi vogliamo eseguire quel programma

**Fino a qui non abbiamo nessuna esecuzione**

## Fasi dinamiche

Dobbiamo decidere quale **architettura concreta** e poi **caricare il programma per quella configurazione (deployment)**

Poi cominciare la **reale esecuzione**, da cui **ottenere dati e la migliore performance attraverso processi in esecuzione**

**Noi siamo interessati alla esecuzione e magari anche a diverse possibili architetture diverse, senza rifare le fasi statiche**

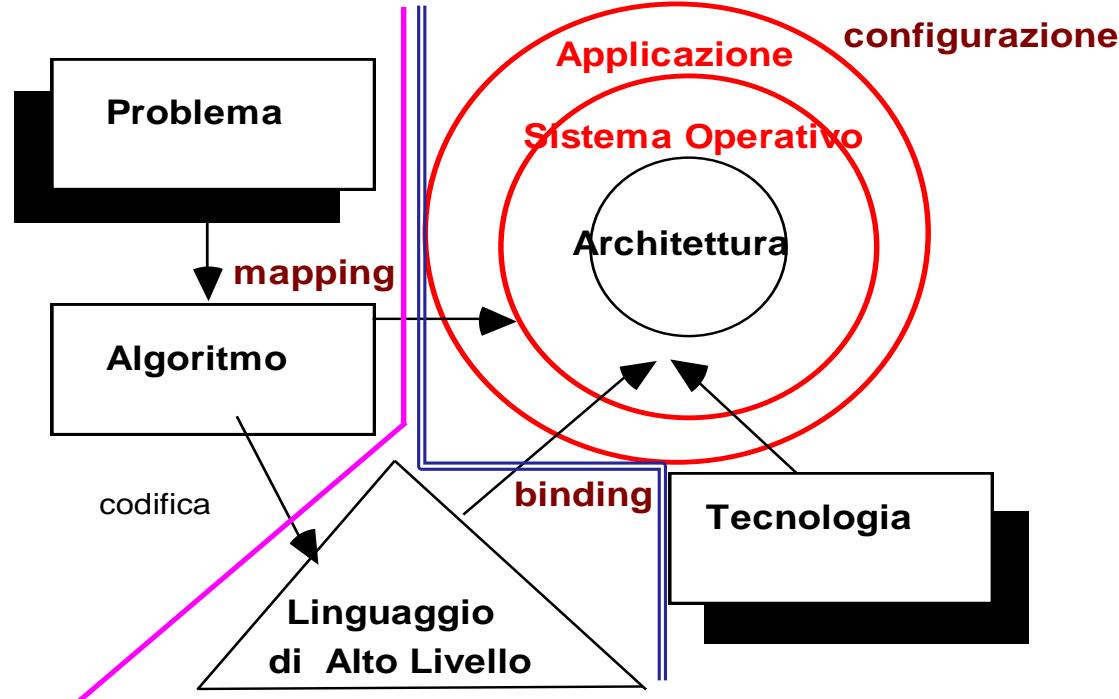
# ARCHITETTURA SUPPORTO

Per una applicazione vorremmo avere fasi dinamiche molto efficienti e con la possibilità di usare al meglio le risorse e di non sprecarle

**Magari molte esecuzioni diverse**

Nel corso massimo interesse per **le fasi dinamiche**

**Deployment ed Esecuzione (molteplici) attraverso processi**



# BINDING delle RISORSE

**Per una applicazione distribuita a processi**

Bisogna potere riferire un'altra **entità** che si trova allocata in un nodo che non è prevedibile a priori e che può cambiare locazione

## GESTIONE BINDING

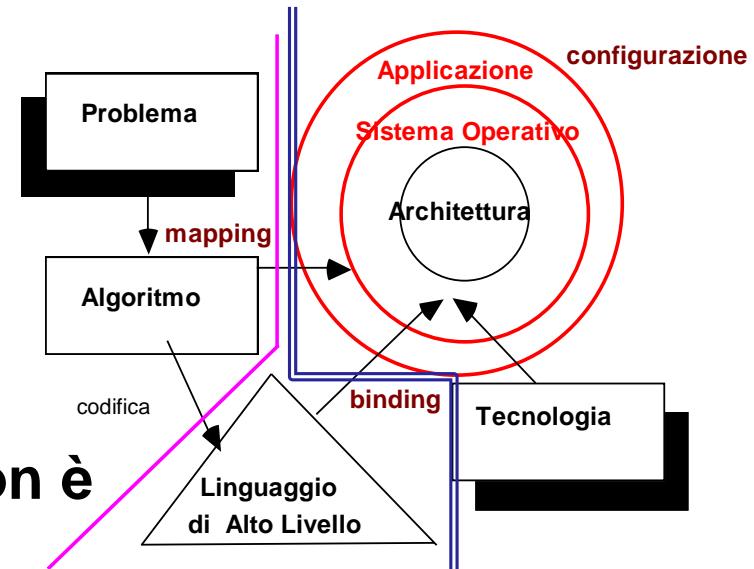
**Consente di potere riferire  
un altro processo con cui dovremo  
Interagire**

## BINDING DINAMICO

In genere, la allocazione delle risorse **non è  
prevedibile prima della esecuzione**

È necessario riuscire a riferire in modo efficiente i processi con cui si vuole comunicare

**SERVIZI di NOMI** che ci consentono di **ottenere le allocazioni  
correnti della entità** necessarie per eseguire



# **INTEGRAZIONE DEI SISTEMI**

---

**Una esecuzione efficiente** dipende da una **comprendizione completa** di tutti i livelli

**Dalla applicazione fino a tutti i livelli del supporto**

**CONSAPEVOLEZZA del SUPPORTO**

**FINO ALLA APPLICAZIONE**

Nel caso di **programmi di sistema** è necessario acquisire una capacità di mettere in relazione i **rapporti tra i diversi livelli** e come i livelli richiedono risorse fino alla interazione con le ricorse locali

**Livello Applicativo**

---

**Livello supporto linguaggio (C, Java, ...)**

---

**Protocolli di comunicazione**

---

**Sistema Operativo**

---

**Risorse locali**

# **Principi di progetto**

---

**Proprietà di base al progetto di sistema**

**Semplicità**

**Efficienza**

**Orientamento ai requisiti**

**applicate a tutti i livelli**

**Processi**

**Strutture dati**

**Algoritmi**

**Risorse e loro controllo**

**Protocolli di interazione**

**Interazione della applicazione e controllo**

# STANDARD PROCESSI

---

Per i **sistemi operativi** esiste uno **standard di processo** per **funzioni di accesso (API)** e modello architetturale **UNIX**

**Sistema operativo concentrato (e relativa macchina virtuale)**  
**modello di conformità**

- caratteristiche di **accesso ai file** (*open/read/write/close*)
- il **progetto di filtri** per una corretta gestione delle **risorse**
- possibilità di **concorrenza** (*fork/exec*)
- possibilità di **comunicazione**

**con definita organizzazione del kernel e API invocabili da diversi ambienti**

**Importanza degli STANDARD, anche di fatto rinforzato da OPEN SOURCE e FREE SOFTWARE**

Tutte le soluzioni più diffuse sono ispirate a questo:

- **Linux** ed altre evoluzioni che fanno ancora i conti con **UNIX**
- Microsoft **Windows OS** che introduce alcuni altri elementi

# UNIX come STANDARD

---

Per i **processi** esiste condiviso il modello a **processi pesanti**, mentre meno accettata è la specifica di processi leggeri

Alcuni **sistemi** si discostano poco per ottenere **migliori prestazioni** nella gestione introducendo processi leggeri

Mantenendo le **funzioni di accesso (API)** e il modello architetturale  
**Esempio: non si usano più kernel monolitici (come primo UNIX)**  
che si accettano in blocco (o meno) che possono produrre overhead  
per modifiche anche minime

**Uso di microkernel (vedi UNIX, WINxx, ... ecc. ecc.)**

realizzazioni minimali del supporto (**meccanismi**) di S.O. nel kernel con  
**le politiche** realizzate a livello applicativo sopra al kernel

- apertura a nuove strategie (**generalità e flessibilità**)
- **costi superiori** delle strategie (rispetto a soluzione nel kernel)

I microkernel contengono supporto ai **processi** e **comunicazione** tra loro, politiche realizzate in spazio utente

# MODELLO dei PROCESSI

---

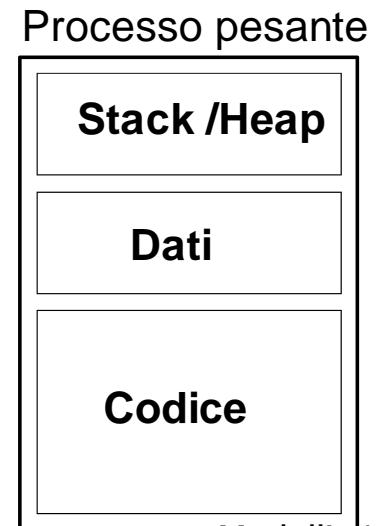
Processi differenziati ⇒ processi **pesanti** / processi **leggeri**

**IMPLEMENTATIVAMENTE**

Processo come aggregazione di parecchi componenti

Uno **SPAZIO di indirizzamento** e uno **SPAZIO di esecuzione**  
insieme applicativo di **codice, dati** (statici e dinamici),  
**parte di supporto**, cioè di interazione con il sistema  
**operativo (file system, shell, socket, etc.)** e per la  
**comunicazione**

I **processi pesanti** richiedono  
molte risorse: ad esempio in UNIX  
il cambiamento di contesto è un'operazione  
molto pesante con overhead  
soprattutto per la parte di sistema



# PROCESSI LEGGERI

**Processi leggeri** ⇒ per ovviare ai limiti dei processi **pesanti** sono definite **entità più leggere** con **limiti precisi di visibilità e barriere e possibilità di condivisione**

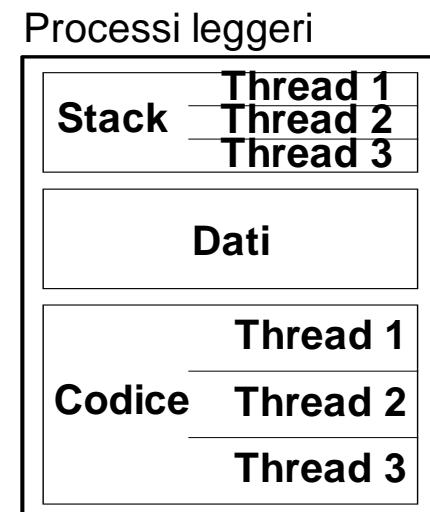
Processi leggeri sono attività che **condividono tra di loro la visibilità di una ambiente contenitore** caratterizzate da uno **stato limitato** e a **overhead limitato**

ad esempio in UNIX, le librerie di thread o in Java i thread  
(purtroppo non si è affermato uno standard)

Il **contenitore unico** dei processi leggeri è un processo pesante che fornisce la visibilità comune a tutti i thread

Tutti i sistemi vanno nel senso di offrire granularità differenziate per ottenere un servizio migliore e più adatto ai requisiti dell'utente

In Java i **thread** sono supportati come **processi leggeri all'interno di un unico processo pesante**



# **STANDARD nel Distribuito**

---

## ***Problema fondamentale***

**Sistema distribuito fatto di nodi anche molto diversi ed eterogenei con esigenze difficili da prevedere tutte prima della esecuzione**

Oltre a **UNIX** come macchina virtuale standard, altri approcci detti... **APPROCCI AD AMBIENTI APERTI**

Uso di un **ambiente aperto unificante** per superare le **differenze delle diverse piattaforme** anche durante la **esecuzione** senza bloccare il sistema e pensare ad un riprogetto

**UNIX ha rappresentato un ambiente molto accettato per alcune aree** (file e comunicazione)

**Nel caso di processi leggeri, la comunità UNIX non ha standardizzato in modo adeguato il modello di threading di processi leggeri**

Sono presenti molte proposte di processi leggeri non portabili e senza fornire garanzie di durabilità del codice

# JAVA come STANDARD

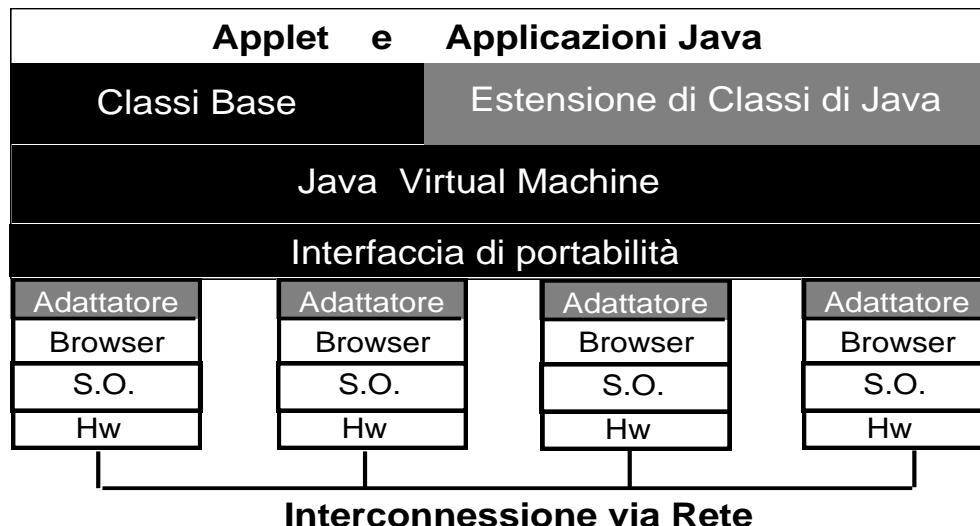
## Java (da JDK1.5)

Linguaggio **Object-Oriented** legato ad un **insieme di librerie** per legarsi e richiedere le **funzioni del sistema operativo nativo** sottostante

- **processi leggeri (thread)**
- **file system** e risorse di sistema
- **risorse di sistema e di comunicazione (Web, sicurezza, ...)**

**Le versioni di Java vanno oltre nella integrazione**

- gestione, monitoraggio, e accounting di risorse
- modello a processi da migliorare ...



In Java, tipicamente la **JVM** è un **processo pesante** che contiene i **thread** che vengono mappati in **processi leggeri interni alla JVM**

# **Principi di sviluppo**

---

**Progetto di applicazioni di due o più entità**

**Design di due tipi di processi, C e S, che devono cooperare per produrre la applicazione**

**Uso di linguaggi Java e C**

**Struttura del processo (interazione utente)**

- Invocazione con argomenti - Controllo argomenti**
- Inizializzazione dello stato**
- Corpo del programma (con I/O) e lavoro spesso come filtro a stream e comunicazione**
- Terminazione (o nessun momento finale del processo)**

# Principi di progetto

---

**Regole di base al progetto di sistema con programmi di dimensioni limitate che eseguiranno moltissime volte**  
**Semplicità**

**La soluzione deve essere ispirata ad impiegare il numero minimo di risorse e avere la massima leggibilità**

**Efficienza**

Partendo da un **progetto più semplice possibile** per piccoli programmi, **dovremmo anche ricercare la massima efficienza (usare variabili statiche e non dinamiche)**

**Orientamento ai requisiti**

**Non progettare soluzioni troppo generali e non richieste**

# ARGOMENTI DI UN PROGRAMMA: ARGC E ARGV

---

Il main prevede due argomenti che definiscono gli argomenti di invocazione del processo (parte di sistema?)

```
int argc;    char ** argv,
```

Sul primo argomento **intero** – è il numero degli argomenti ,  
nome programma incluso

Il **secondo** è un **puntatore di puntatori a carattere** ossia un array  
aperto di puntatori a carattere  
lo leggiamo

```
char * (* argv)
```

**ARG PUNTATORE A PUNTATORI A CARATTERI**

# ARGV

ARGV come puntatore di puntatori a carattere  
Non matrice di caratteri in sequenza

char \*\* argv; qui argc 5

char \* (\* argv)

argv

argv[0]

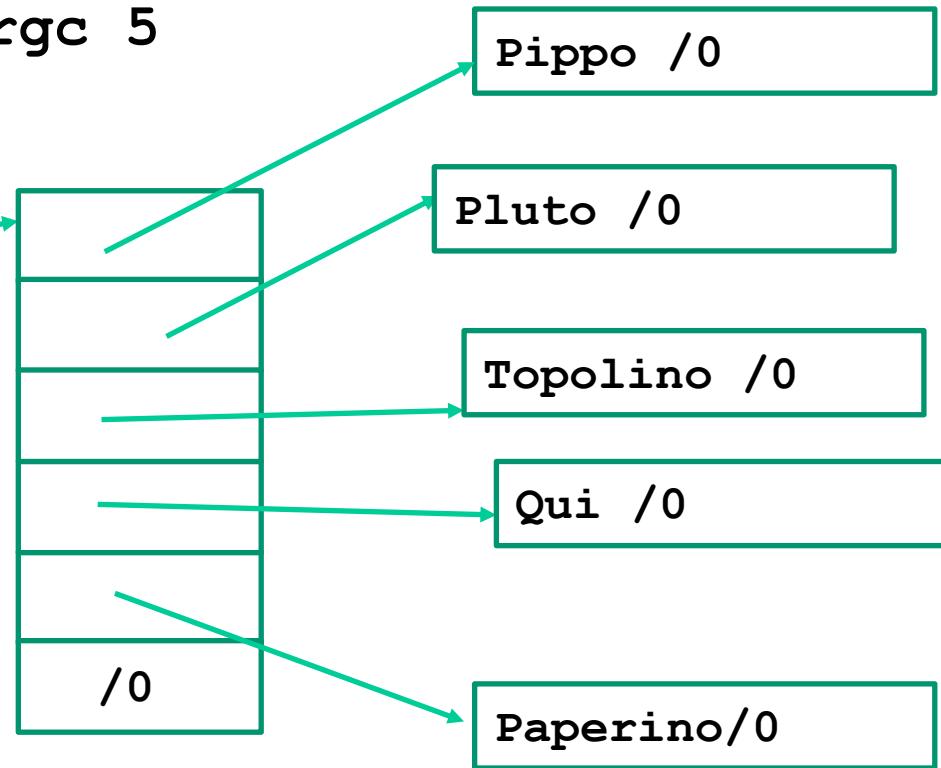
argv[1] - argv+1

argv[2] - argv+2

argv[3] - argv+3

argv[4] - argv+4

argv[5] - argv+4



L'array termina con 0 binario (dopo argc valori)

# ARGV

---

**ARGV non è una matrice argomenti    ARGS  
matrice di caratteri in sequenza**

`char argv [5][6];`

**5 righe e 10 colonne**

**Gli array e le matrici sono costanti (non puntatore a costanti)**

`argv[0][1]`

`argv[4][1]`

`argv[0]`

`argv[3]`


**Solo 5 righe di caratteri**

**Nessun terminatore e nessuna stringa**

# TRATTAMENTO ARGOMENTI IN C ARGV

---

**Controllo argomento intero**

```
char ** argv;
```

**Per argv[1] che sia intero e ricavarne valore intero**

```
j=0;  
while (argv [1][j])  
    if ((argv[1][j]<'0') | (argv[1][j]>'9')) {exit...}  
valore = atoi(argv[1]);  
...
```

**Ogni stringa termina con uno 0 binario**

**atoi converte in intero tutto il pattern di cifre intere che trova  
nella stringa**

# File

---

Un **file** è una **sequenza di dati, di dimensione grandi a piacere (idealmente infinita)**

Questa **risorsa NON SI può tenere tutta in memoria**

**Alcuni fatti sui file:**

- **I file sono flussi o sequenze di byte (stream)**
- **Alcuni file sono testo:** ossia contengono **solo caratteri ASCII** (e non byte qualunque) e sono **fatti di linee di caratteri** (di dimensione limitata), ossia sequenze di caratteri ASCII inframezzate da fine linea ('\n'), se ben fatti
- **I file sono risorse non trasportabili tra processori, perché sono radicate sul disco su cui risiedono;** si può trasportarne il contenuto con opportune applicazioni che possiamo codificare con algoritmi ad-hoc

# File in UNIX

---

**Il modello è quello di UNIX**

Un file come **risorsa che fa riferimento alla memoria permanente di un nodo**

**Un file è memorizzato su disco ed è legato al macchina su cui risiede il disco di residenza del file**

**L'accesso ai dati avviene con primitive di sistema**

Corso di RETI

**Il file** è un componente della macchina di residenza, ossia **una risorsa di sistema** come i **dispositivi**, le **driver di comunicazione**, le **configurazioni locali**, ...

**NON HA SENSO TRASFERIRLI DA UNA MACCHINA AD UNA ALTRA CON UNA OPERAZIONE DI LINGUAGGIO...**

**I linguaggi non hanno primitive per farlo**

# Modello dei File in UNIX

---

Dettagli sui file

Il file prevede **un descrittore che ne dice la lunghezza, e non contiene una fine del file (un carattere o una sequenza ad-hoc)** e un contenuto in byte

- I **file possono essere generali (file binari) o limitati in contenuto**
- I **file testo** sono file **che contengono solo caratteri ASCII** e sono costituiti da **linee di dimensione limitata** (ossia caratteri ASCII e fine linea '\n')

Quando interagiamo con il file, **in lettura, le azioni primitive sui file hanno il modo di farci capire che siamo arrivati alla fine del file** e non possiamo leggere ulteriori caratteri, **tramite risultati specifici che ci fanno capire** che abbiamo raggiunto la fine del file (la condizione **EOF non è una eccezione**)

# Primitive sui file e dispositivi

---

In generale, il modello di lavoro è quello di azioni richieste e attuate dal sistema di supporto

Il sistema per le azioni mette a disposizione **operazioni primitive** sui file e dispositivi che sono visti come **stream esterni ai linguaggi**

**Il modello di interazione è quello detto**

**Open, Read / Write, Close**

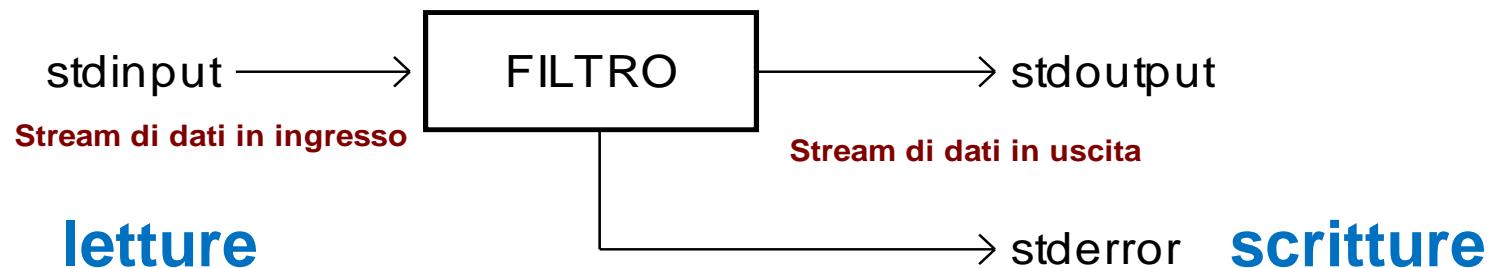
che agiscono **in modo atomico sugli stream** e non in modo **mediato da una libreria** (vedi API precedenti)

Con altre azioni primitive, possiamo anche intervenire sui dispositivi e gestirli e controllarli in modo più granulare

**(select(), ioctl(), fctl(),....)**

# Filtrari (ridirezione e piping)

Un **filtro** è un modello di ordinato e ben fatto che prevede un **programma che riceve in ingresso da un input e produce risultati su output (uno o più)**



Il **filtro deve consumare tutto lo stream in ingresso e portare in uscita il contenuto filtrato**  
**Possibilità di ridirezione e di piping**

```
comando > file1 < file2  
rev < file1 | sort | tee file | rev | more
```

# Input/Output in C

---

C non definisce l'input/output ma lo prende dal **sistema operativo**, che prevede una **gestione integrata di I/O e dell'accesso ai file**

**Per I/O, azioni di base del Sistema Operativo:**  
**primitive** di accesso **read()** / **write()**  
**con semantica di atomicità (ossia di mutua esclusione e di non interruzione)**

**Per I/O, anche libreria:**

Input/Output a caratteri

**getc()**, **putc()**

Input/Output a stringhe di caratteri

**gets()**, **puts()**

Input/Output con formato per tipi diversi

**scanf()**, **printf ()**

Input/Output con strutture FILE

**fopen()**, **fclose ()**

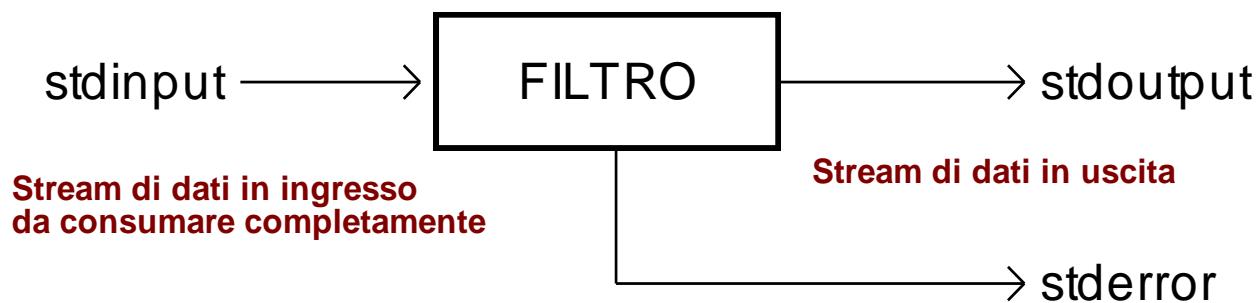
# I/O azioni e API

**L'input è sempre più complesso dell'output**

Si deve **gestire la fine dello stream di Input**, ossia si deve gestire la **fine del file (che è un evento atteso e necessario, non una eccezione)** e **consumare sempre tutto l'input**

**I programmi che non lo fanno sono scorretti e non possono essere usati facilmente in composizione**

Per questo progettiamo sempre **programmi filtri (e processi) corretti** che devono tenere conto delle risorse in gioco e favorirne la gestione con sequenze di azioni opportune (API)



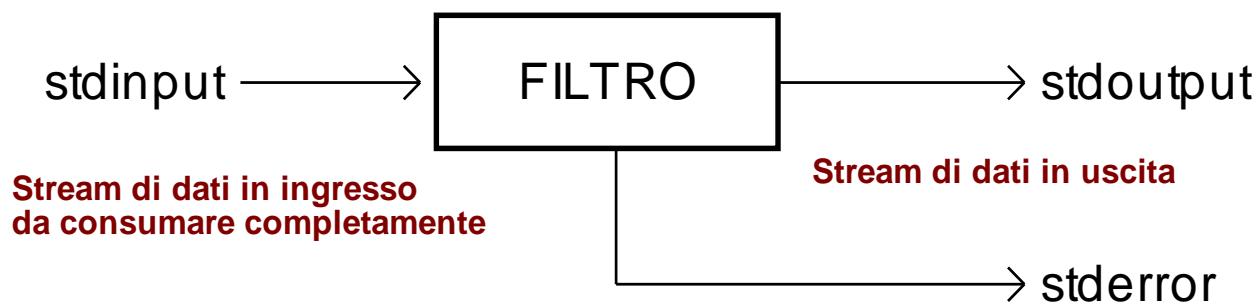
# Input: protocollo

L'input è uno stream di dati (spesso file testo, ossia costituito da soli caratteri ASCII con fine linea, come quelli che derivano da tastiera)

Spesso i file che vengono dati come input sono di questo tipo  
**Protocollo di input**

Il processo deve leggere l'input secondo le necessità applicative (a caratteri, a blocchi di caratteri, a sequenze) tipicamente in modo iterativo...  
**fino a non incontrare la fine dello stream**

**Il non consumo di parte dello stream è assolutamente scorretto  
specie in catene o in composizione di processi**



# Funzioni di I/O a caratteri ASCII

---

**int getchar(void);** legge un carattere e **restituisce il carattere letto convertito in int o EOF in caso di end-of-file o errore**

**int putchar(int c);** scrive un carattere e restituisce il carattere scritto o **EOF in caso di errore**

**Esempio: Programma che copia da input ad output:**

```
#include <stdio.h>

main()
{
    int c;

    while ((c = getchar()) != EOF)    putchar(c);
}
```

**ATTENZIONE:** La funzione getchar() (a causa della driver) comincia a restituire caratteri solo quando è stato battuto un carriage return (invio) e il sistema operativo li ha memorizzati

**L'evento di terminazione è EOF == -1**

# Funzioni di I/O a stringhe

---

**Obiettivo: fornire e lavorare con stringhe ben fatte**

- le stringhe di caratteri vengono memorizzate in array di caratteri terminate dal carattere '\0' (NULL - valore decimale zero)

**char \*gets (char \*s);** legge una stringa e restituisce la stringa come indirizzo del primo carattere, se ok; **in caso di end-of-file o errore ritorna stringa nulla (0 ossia carattere NULL)**

**int puts (char \*s);** scrive una stringa, in caso di errore restituisce EOF

Esempio (lo stesso di prima):

```
#include <stdio.h>  
main() { char s[81];  
while (gets(s)) puts(s); }
```

- gets sostituisce **new line** con NULL, puts aggiunge **new line** alla stringa

**Le gets non dà problemi se si leggono caratteri all'interno della memoria fornita dall'utente (se a livello di Sistema Operativo si garantisce terminazione corretta, nessun problema)**

# I/O a stringhe: gets e fgets (NON USARE fops)

---

`char * gets (char *s);`

`int puts(char *s);`

legge una stringa e **non controlla** che la memoria sia congrua (OVERFLOW) in caso diamo troppi caratteri

`char* fgets (char *s, int #car, FILE *f);`

`char* fgets (char *s, int #car, stdin);`

legge una stringa dal file specificato, se errore o EOF, restituisce NULL

Se non forniamo il fine linea o terminiamo prima della dimensione specificata, la fgets riceve un carattere di meno e inserisce il carattere NULL all'ultima posizione (inserisce anche il '/n')

In caso di situazioni **in cui il sistema operativo controlli la massima sequenza possibile e tenendo conto di quella, la gets non è problematica**

# Funzioni di I/O con formato

---

Si forniscono funzioni per la lettura/scrittura di dati formattati di ogni tipo con un numero variabile dei parametri (stringhe di formato tra "%")

**int printf (char \*format, expr1, expr2, ..., exprN);**

scrive una serie di valori in base al format, restituendo il numero di caratteri scritti, oppure **EOF** in caso di errore

**int scanf (char \*format, &var1, &var2, ..., &varN);**

legge una serie di valori in base alle specifiche contenute nella format memorizzando i valori nelle variabili passate per riferimento, e restituendo **il numero di valori letti e memorizzati**, oppure **EOF in caso di end-of-file**

(qui **EOF == -1, se inferiore ai valori attesi, problema**)

Esempi:

```
int k;
```

```
scanf ("%d", &k) ;
```

```
printf ("Il quadrato di %d e' %d", k, k*k) ;
```

# Formati comuni (e %)

---

signed int	<b>%d</b> %hd %ld	unsigned int	<b>%u</b> (decimale) %hu %lu
ottale	<b>%o</b> %ho %lo	esadecimale	<b>%x</b> %hx %lx
float	<b>%e</b> %f %g	double	<b>%le</b> %lf %lg
carattere singolo	<b>%c</b>	stringa di caratteri	<b>%s</b>
puntatori (indirizzi)	<b>%p</b>		

Per l'output dei **caratteri di controllo** si usano: '\n', '\t', etc.

Per l'output del carattere '%' si usa: %%                   \% non funziona!

```
int a;  
printf("Dai un carattere e ottieni il valore dec. \\\nottale e hex "); a = getchar();  
printf("\n%c vale %d in decimale, %o in ottale e %x in \\hex.\n",a, a, a, a);  
}
```

# Errore lettura in scanf

---

**scanf**, in caso di errore di formato di lettura, ritorna una indicazione del **numero di variabili correttamente lette** e non **consuma lo stream di input** e **lascia il puntatore di I/O fermo** ad una **posizione iniziale** e ...

Quindi il valore sbagliato rimane nell'input e deve essere **invece consumato**, specie in caso di ciclo di letture (tipicamente fino a fine linea)

Lettura di un intero (gestendo errore)

```
while ((ok = scanf ("%i", &num1)) != EOF /* finefile */ )  
{if (ok != 1) // errore di formato  
 do {c=getchar(); printf("%c ", c);} while (c != '\n');  
printf("Inserisci valore intero, EOF per terminare: ");  
} ...
```

In alternativa la può consumare i caratteri **gets(str)** ;

In genere **scanf non consuma il fine linea** (con formati interi, float, ...) che va tolto **manualmente dall'input**

# Primitive sui file e dispositivi

---

In generale, sono preferibili le **azioni primitive** sui file che agiscono **in modo atomico sugli stream** e non in modo *mediato da una libreria* (vedi API precedenti)

**int read (int fd, char \* buff, int len)**

legge i caratteri e ritorna il numero di caratteri letti. In caso di errore, restituisce un valore negativo. In caso **di fine file restituisce 0**

**int write (int fd, char \* buff, int len)**

scrive i caratteri e ritorna il numero di caratteri scritti. In caso di errore, restituisce un valore negativo.

Con le azioni primitive, possiamo anche intervenire sui dispositivi e gestirli e controllarli in modo granulare (select(), ioctl(), fctl(),....)

Le azioni mediate attraverso le strutture dai **FILE** sono meno dirette e agiscono appunto attraverso **la libreria del sistema operativo**

**fopen (), fread (), fwrite (), fgets (), fputs (), fscanf (), fprintf (), fclose()**

# C COME LINGUAGGIO DI SISTEMA

---

In C ogni **variabile o tipo** prevede e mostra **una lunghezza in memoria**

```
short s; int argc, i; float f;  
char *p1; int *p2; char ** argv; sockaddr_in * sa;  
...
```

Di ogni **variabile** possiamo chiedere la dimensione **sizeof**

<b>sizeof(short)</b>	-	<b>sizeof (s)</b>	<b>2</b>
<b>sizeof(int)</b>	-	<b>sizeof (i)</b>	<b>4</b>
<b>sizeof(sa)</b>	-	<b>sizeof (p1) - sizeof (p2)</b>	<b>4</b>
<b>sizeof(argv)</b>	-		<b>4</b>

Ogni puntatore rappresenta un **indirizzo** e quindi sono tutti interscambiabili

```
p1=p2; p2=sa; p2 = argv;
```

Spesso usiamo **cast solo per leggibilità (no codice espanso)**

```
p1= (char*)p2;
```

# JAVA COME LINGUAGGIO DI SISTEMA

---

**Java non è un linguaggio di sistema, ma è un linguaggio ad oggetti per fare applicazioni**

- Promuove una programmazione molto sicura (**safe o corretta**)
- Nasconde la memoria e non ne permette un facile accesso
- Tende a nascondere il formato dei dati che è supportato dalla JVM e non visibile
- Non fornisce strutture flessibili o metodi di basso livello, volendo fare strutture molto chiare dei tipi di dato e dei dati
- In Java non sono visibili gli indirizzi (nascosti) anche se la JVM ha una semantica per riferimento tra oggetti ('semantica per riferimento' tra oggetti) in una JVM
- In Java possiamo scrivere programmi che diventano processi (args, I/O, file, ...)

# JAVA: VARI TIPI DI COSTRUTTI

---

**Cosa è una Interfaccia a confronto con una Classe?**

**Sono entrambi descrittori di istanze**

**Uno concreto (con metodi), l'altro astratto**

**Ereditarietà**

**Relazione di derivazione di comportamenti da altre entità dello stesso tipo: chi eredita ha tutto quello specificato dalla super entità**

**Tra classi ereditarietà**

**singolo genitore**

**Tra interfacce**

**ereditarietà multipla**

# RMI: PRODROMO

Che entità / oggetti ci sono in Java? A run-time

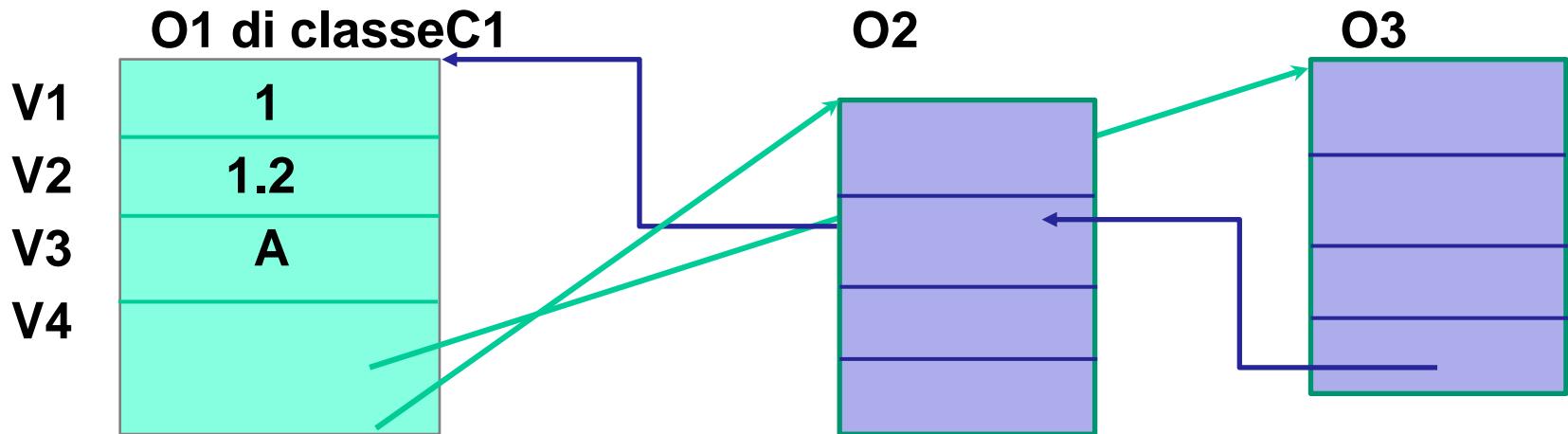
Classi? Istanze? Interfacce? tutti

Come sono le istanze? Semantica per riferimento

Una istanza contiene le variabili descritte dalla classe e riferisce i metodi nella classe (valori per primitivi e riferimenti per altri oggetti)

Gli oggetti non sono contenuti dentro gli oggetti ma si puntano tra loro creando un grafo per ogni oggetto (semantica locale per riferimento)

Un oggetto richiede di vedere anche tutti gli oggetti puntati



# JAVA: PRODROMI DI SISTEMA

---

Interfaccia vs Classe

sono entrambe entità di metalivello, ossia descrittive delle istanze

Ogni istanza riferisce una ed una sola classe (da cui è creata)

Una classe implementa molte interfacce

Tipi di variabili nelle istanze (a parte variabili di classe)

Variabili nelle istanze (primitivi o riferimenti ad istanze)

Variabili interne ad un oggetto istanza sono tipizzate (con tipi primitivi o classi)

Variabili interfaccia (tipizzate dalla interfaccia)

Variabili interfaccia possono puntare ad una istanza di una classe che implementi la interfaccia stessa

# JAVA: INPUT E OUTPUT

---

**Gli oggetti di Java vivono nel contesto degli oggetti  
(molto tipato e controllato) dalla JVM che fa da controllore e  
gestore**

**Si possono anche avere File e Input Output  
Con opportuno supporto**

**Come facciamo a vedere il contenuto di un oggetto per  
cavarne il contenuto?**

# JAVA: INPUT E OUTPUT

---

**Gli oggetti di Java vivono nel contesto degli oggetti  
(molto tipato e controllato) dalla JVM che fa da controllore e  
gestore**

**Si possono anche avere File e Input Output  
Con opportuno supporto**

**Come facciamo a vedere il contenuto di un oggetto per  
capiro il contenuto?**

**Dobbiamo farne una serializzazione che porta l'oggetto in un  
formato a byte ‘visibile’ ed esterno, salvabile sul disco, ed  
eventualmente ripristinabile nella macchina virtuale**

# LA SERIALIZZAZIONE

---

i dati contenuti in un **oggetto Java** possono essere scritti e letti (dal disco) attraverso una **serializzazione /deserializzazione, ossia inseriti in un messaggio in sequenza o in modo lineare**, utilizzando le funzionalità offerte **direttamente a livello di supporto al linguaggio (Interface Serializable)**

- **Serializzazione**: trasformazione di oggetti complessi in **semplici sequenze di byte** (interi grafi di oggetti)
  - metodo `writeObject()` su uno stream di output
- **Deserializzazione**: decodifica di una sequenza di byte e costruzione di una copia dell'oggetto originale
  - metodo `readObject()` da uno stream di input

# JAVA: PROCESSI E DINAMICA

---

**Gli oggetti di Java vivono nel contesto degli oggetti  
(molto tipato e controllato) dalla JVM che fa da controllore e  
gestore**

**Si possono anche avere Processi e esecuzione dinamica  
Con opportuno supporto**

**Come facciamo a produrre esecuzione in Java?**

**Siamo all'interno del processo pesante della JVM e abbiamo  
solo thread che condividono i dati della JVM**

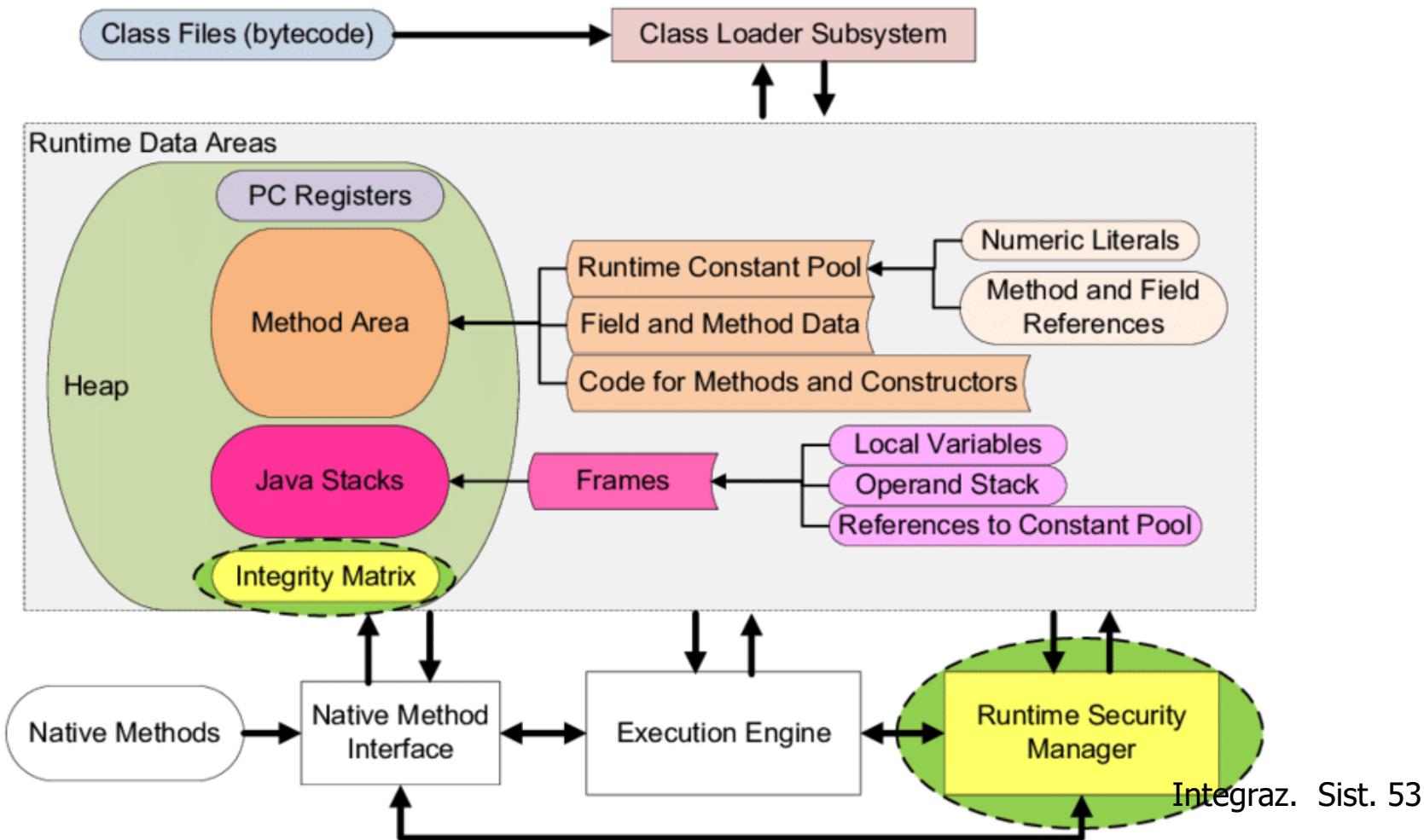
**Ogni thread ha il suo stack separato**

**Ogni oggetto (o classe) è caricato dinamicamente al bisogno  
nella memoria heap dinamica**

**l'heap della JVM ospita tutti gli oggetti, che sono allocati e  
deallocati**

# VISIONE INTERNA JVM JAVA

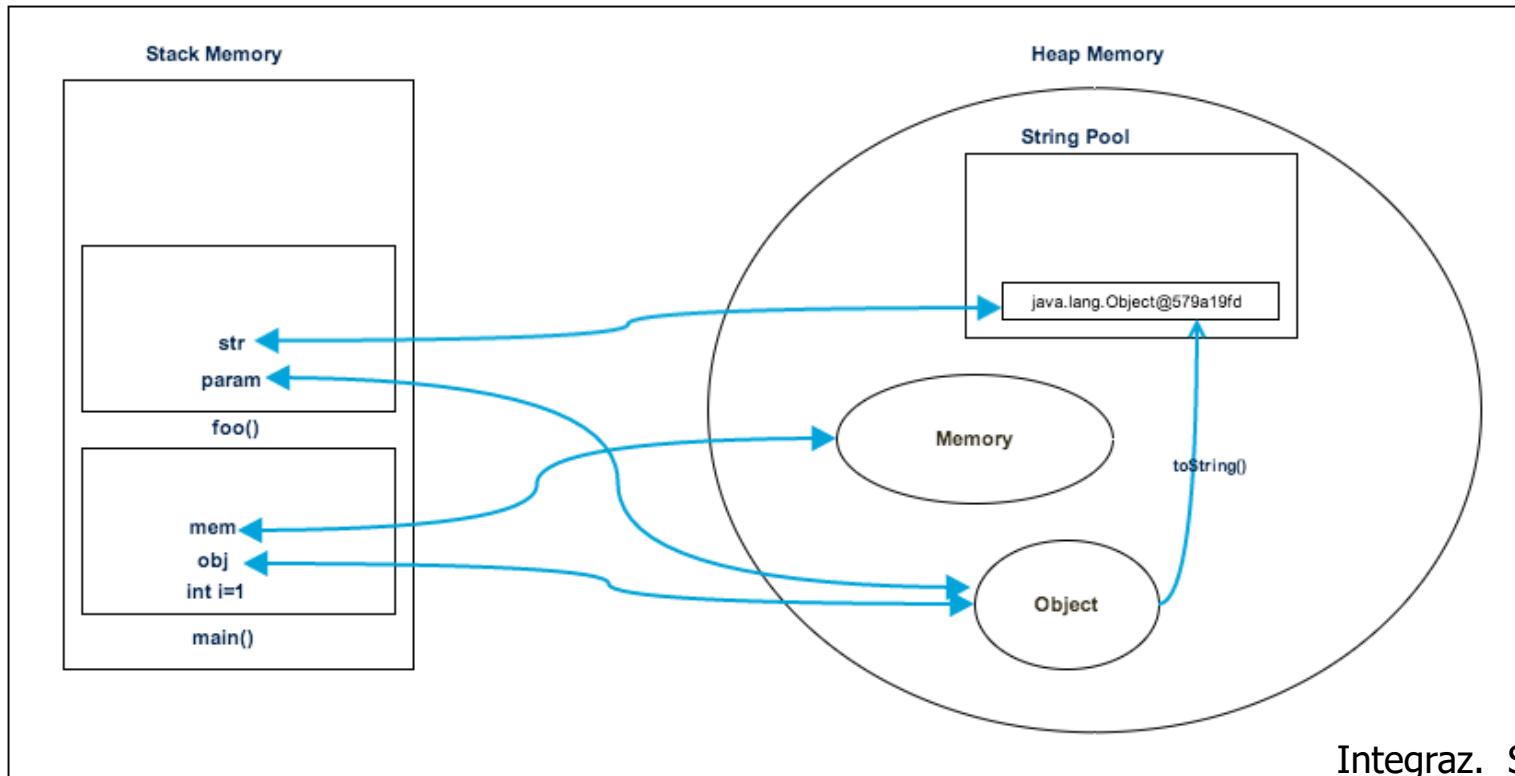
La memoria della JVM è tipicamente caricata al bisogno (e non in modo statico preventivo): le istanze e le classi sono caricate solo quando sono effettivamente necessarie alla esecuzione



# VISIONE DELLA MEMORIA DELLA JVM

La memoria in Java si basa su Heap nella cui memoria si mettono tutti gli oggetti (istanze e classi) allocati dinamicamente

I singoli stack (uno per ogni thread) riportano le copie degli oggetti (con le variabili istanza e interfaccia interne che tipicamente puntano agli oggetti nell'heap)



# ARCHITETTURA SUPPORTO

---

**Una esecuzione efficiente richiede anche una abilità nella parte di progetto per avere componenti di cui si possano fare deployment efficienti in ogni ambiente**

Nel caso di programmi di **integrazione di sistema** (non applicativi) che devono essere **parte del supporto** e magari **eseguiti milioni di volte**, allora il progetto deve mirare a:

- **Semplicità delle strutture dati e degli algoritmi**
- **Minimizzazione del costo della configurazione**
- **Minimizzazione dell'impegno sulle risorse**
- **Minimizzazione overhead**
- **Capacità di interazione con l'utente**
- **Prevenzione errori**
- **Capacità di controllo della esecuzione**
- ...

# SEMPLICITÀ DELLE STRUTTURE DATI

---

## Semplicità delle strutture dati

### - Strutture statiche

Le variabili in memoria **non dinamica** sono predefinite e non hanno costi aggiuntivi durante la esecuzione (overhead della allocazione dinamica delle aree sullo stack e dall'heap)

### - Struttura dati limitate

**Le variabili con memoria limitata (array)** non hanno costi in eccesso per memoria non usata (allocazione di liste linkate)

### - Strutture dati semplici e non troppo astratte

**Le strutture dati semplici** (array e non lista linkata o hash list o hash table) non hanno **capacità espressiva non richiesta**

### - Controllo delle risorse allocate

### - Prevenzione errori

- Capacità di controllo della esecuzione

- Capacità di interazione con l'utente

# SEMPLICITÀ DELLE STRUTTURE DATI (1)

Vediamo come cambiano le performance in base **alla complessità delle strutture** utilizzate per implementare un algoritmo in C

**ESEMPIO a:** Scrivere un programma che memorizza una serie di interi (100.000) e verificarne le performance

```
startTime = System.currentTimeMillis();
for (int i=0; i<99999; i++) arr[i]=i+1;
stopTime = System.currentTimeMillis(); elapsedTime = stopTime - startTime;
System.out.println("Fill Simple Array: "+elapsedTime + ", start="+startTime+
", stop="+stopTime);
/*Fill Simple Array: 2 millisecondi, start=1536829145957,
stop=1536829145959*/
```

```
startTime = System.currentTimeMillis();
for (int i=0; i<99999; i++) arrL.add(i+1);
stopTime = System.currentTimeMillis();
elapsedTime = stopTime - startTime;
System.out.println("Fill Array List: "+elapsedTime + ", start="+startTime+
", stop="+stopTime);
/*Fill Array List: 18 millisecondi, start=1536829145960,
stop=1536829145978*/
```

# SEMPLICITÀ DELLE STRUTTURE DATI (2)

Vediamo come cambiano le performance in base alla **complessità delle strutture dati**

**ESEMPIO b:** Scrivere un programma che conti tutte le occorrenze di un valore in un struttura

```
startTime = System.currentTimeMillis();
for (int i = 0; i < arr.length; i++) if (arr[i]==5000) occ++;
stopTime = System.currentTimeMillis();
System.out.println("Occurrences = "+occ);
elapsedTime = stopTime - startTime;
System.out.println("Read Simple Array "+elapsedTime + ", start="+startTime+
", stop="+stopTime);
/*Read Simple Array 2 millisecondi,
start=1536829938252, stop=1536829938254*/
```

```
startTime = System.currentTimeMillis();
System.out.println("Occurrences = " +
    java.util.Collections.frequency(arrL,5000));
stopTime = System.currentTimeMillis();
elapsedTime = stopTime - startTime;
System.out.println("Read Array List "+elapsedTime + ", start="+startTime+",
stop="+stopTime);
/*Read Array List 11 millisecondi,
start=1536829938254, stop=1536829938265*/
```

# SEMPLICITÀ DEGLI ALGORITMI (E DATI)

---

## Semplicità degli algoritmi

### - Algoritmi strutturati

In generale, e più specificatamente, se procedure e funzioni sono invocate una volta sola, fare codice in linea (**NON passare attraverso catene di invocazioni di procedure innestate, se si può evitare**)

### - Strutture con controllo semplici

**Usare sequenza, alternative e ripetizioni semplici in modo ordinato ed evitare i break**

Si faccia uso di cicli `while (condizione) {...}`

### - Definizione dei dati

Le strutture dati devono essere **allocate una volte per tutte e non nei cicli per evitare costi aggiuntivi e non necessari**

### - Controllo delle risorse allocate

### - Prevenzione errori

### - Capacità di controllo della esecuzione

### - Capacità di interazione con l'utente

# SEMPLICITÀ DEGLI ALGORITMI (1)

## Semplicità degli algoritmi

In generale, soprattutto se procedure e funzioni sono invocate una volta sola, fare codice in linea

Una funzione ad ogni passo

```
int function(int num1, int num2) {  
    if (num1 % 2 == 0)  {return sqrt(num2);}  
    else { return (int)pow(num1, 2);}  
}  
  
int main(int argc, char* argv[]) {  
    int a = atoi(argv[1]);  
    for (int i = 0; i < 1000000000; i++) { int res = function(i, a); }  
}
```

Tempo di esecuzione totale  
22.2 secondi

Un programma unico

```
int main(int argc, char* argv[]) {  
    int a = atoi(argv[1]);  
    int res = 0;  
    for (int i = 0; i < 1000000000; i++) {  
        if (i % 2 == 0)  {res = sqrt(a);}  
        else { res = (int)pow(i, 2); }  
    }  
}
```

Tempo di esecuzione totale  
6.7 secondi

# SEMPLICITÀ DEGLI ALGORITMI

---

## Semplicità degli algoritmi

### - Algoritmi semplici

Eliminare **controlli ridondanti** per ridurre i costi in eccesso  
(vedi funzioni sulle stringhe in C che **non** controllano formato)

### - Poco innestamento di procedure/funzioni non necessarie

Eliminare l'impegno **sullo stack** che viene anche da innestamento funzioni (ogni invocazione impegna il record di attivazione sullo stack)

### - Eliminare ripetizioni

Uso di **variabili per tenere lo stato di una esecuzione** invece di ripetere la computazione più volte, una volta per necessità (array che viene scorso una prima volta per contare occorrenze e poi una seconda volta viene scandito ogni volta per fare una azione per ciascuna di queste – pensare a 100.000 elementi e magari milioni di volte o più)

### - Capacità di controllo della esecuzione

### - Capacità di interazione con l'utente

- ...

# SEMPLICITÀ DEGLI ALGORITMI (2)

- Prendiamo come esempio l'algoritmo che calcola la sommatoria, realizzato in modo **ricorsiva** vs **iterativo**

```
int sommatoriaRic(int i)
{   if(i!=0)
    return i+sommatoriaRic(i-1);
else return i;
}
```

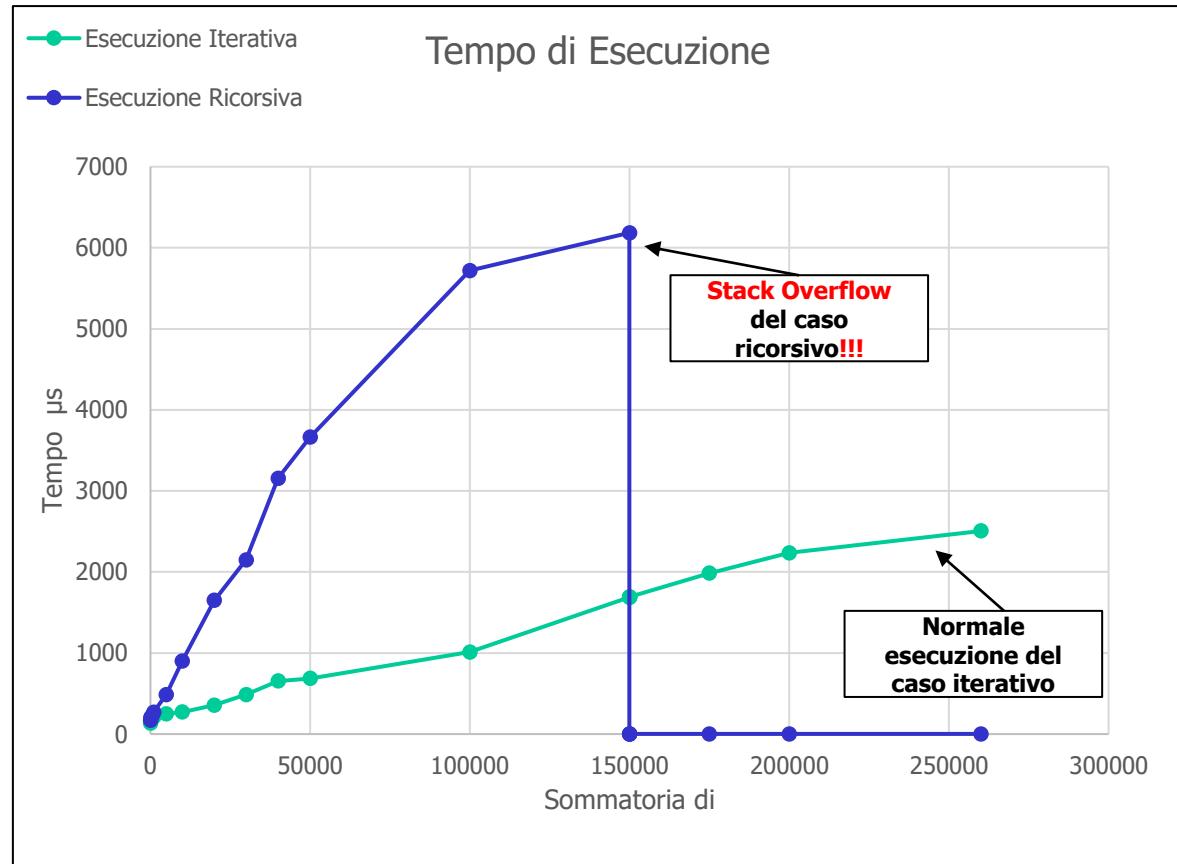
```
int sommatoriaIt(int i)
{ int acc = 0, j=0;
for(j=i; j>0; j--)
    acc=acc+j;
return acc;
}
```

- Esiste un valore per il quale le due funzioni si comporteranno in maniera diversa: cosa succede alla versione **ricorsiva**?

# SEMPLICITÀ DEGLI ALGORITMI (2)

Stampa dei risultati:

- Eseguendo diversi test durante i quali è stato aumentato il valore per il quale si deve calcolare la sommatoria, si ottiene la situazione del **grafico a lato**
- Oltre ai tempi, possiamo notare come il **caso ricorsivo riempie la risorsa stack** molto prima di quello iterativo, *a parità di valori da calcolare*



# ESEMPIO di DATI di SISTEMA (3)

## La tabella dei file descriptor è statica

Scrivere un programma che mostri il numero massimo di descrittori di I/O che il processo può creare

Ad ESEMPIO:

```
while(true)
{
    printf("Opening file n. %d ... \n", i);
    if((open("test.txt", O_WRONLY|O_CREAT|O_TRUNC, 00640))<0)
        { perror("Error opening file \n"); exit(1); }
    i++;
}
```

- Arrivati al limite dei descrittori il sistema lancia ‘Too many open files’

Una volta eseguite le operazioni necessarie su un file aperto, è fondamentale chiudere il file per liberarne il descrittore

```
while(true)
{
    printf("Opening file n. %d ... \n", i);
    if((fileDescr=open("test.txt", O_WRONLY|O_CREAT|O_TRUNC, 00640))<0)
        { perror("Error opening file \n"); exit(1); }
    /*... operations on file ...*/
    printf("Closing file ... \n");
    close(fileDescr); i++;
}
```

# Funzioni in C per stringhe

---

**Il C non definisce le funzioni per I/O, ma, dove possibile, le delega al sistema operativo per maggiore flessibilità**

le strutture dati sono definite dal linguaggio, ma le funzioni sono lasciate al sistema operativo o alle librerie

**Obiettivo: lavorare con stringhe sequenze di caratteri**

- le stringhe sono accedute tramite puntatori (char \*)
- le stringhe di caratteri vengono memorizzate in **array di caratteri terminate dal carattere '\0'** (NULL - valore decimale zero)

Tutta una serie di funzioni su stringhe

**int strlen(const char \*s);    char \*strcat(char \*dest, const char \*src);**

**int strcmp(const char \*s1, const char \*s2);**

**char \*strcpy(char \*s1, const char \*s2);**

**char \*strstr(const char \*ss, const char \*s);**

Le funzioni non **controllano le terminazioni** per efficienza e funzionano correttamente solo per stringhe ben fatte

# USO MINIMO DELLE RISORSE

---

## Minimizzazione dell'impegno sulle risorse

Nel caso di **programmi di sistema che devono essere ripetuti molte / moltissime volte** è molto importante avere il **controllo delle risorse usate**

- Non ripetere **operazioni semplici** (usare variabili di stato come si fa con **stile imperativo**)
- Non **allocare e deallocare variabili all'interno di cicli, ma definirle all'esterno del ciclo**
- Non **lasciare memoria non utilizzata**
  
- Considerare bene i processi
- Capacità di controllo della esecuzione
- Capacità di interazione con l'utente

# USO MINIMO DELLE RISORSE

- **ESEMPIO** **allocazione di memoria**: scrivere un programma che riempia lo spazio di Heap allocando dati (senza eseguire mai una **free**...)

```
struct Struttura { int a, b;};

int main(void) {
    struct Struttura *b;
    srand(time(NULL));
//random num gen
    for (;;)
    { b = malloc(sizeof *b);
        b->a=rand(); b->b=rand();
    }
    return 0;
}
```

```
struct Struttura { int a, b;};
int main(void) {
    struct Struttura *b;
    srand(time(NULL));
    for (;;)
    { free(b);
        b = malloc(sizeof *b);
        b->a=rand(); b->b=rand();
    }
    return 0;
}
```

Se vogliamo che il programma **non** fallisca dobbiamo **liberare memoria prima di allocarla** di nuovo

- Usare **dmesg** per verificare la memoria occupata dal processo:

**Out of memory**: Kill process 5208 (o) score 803 or sacrifice child

Killed process 5208 (o) **total-vm:14224868kB**, anon-rss:7300564kB, file-rss:0kB, shmem-rss:0kB

# USO DELLE RISORSE PROCESSI

---

## Minimizzazione dell'impegno sulle risorse

Nel caso di processi e programmi che interagiscono con l'utente

- Controllare gli **argomenti di invocazione** e procedere solo se corretti
- Verificare gli **input dell'utente** e non procedere **in caso di errore** (per non fare azioni inutili e costose)
- Consumare gli **stream (di input)** e le risorse in modo corretto
- Interagire **usando risorse di sistema limitate**
- Progettare bene i processi
- Capacità di controllo della esecuzione
- Capacità di interazione con l'utente
- ...

# USO DELLE RISORSE PROCESSI (1)

- **ESEMPI**

- **Controllo Parametri:** per non avere incongruenze durante l'esecuzione di un programma occorre sempre controllare **NUMERO**, **TIPO** e **RANGE** consentito dei parametri inseriti dall'utente
- Per esempio scrivere un programma che accetti come input la dimensione **INTERA** del buffer da usare durante il trasferimento di un file

```
int num, dimBuffer=0;  
if(argc!=2) // controllo numero argomenti  
{ printf("Usage Error: %s DimBuffer\n", argv[0]); exit(1); }  
...  
//CONTROLLO SECONDO PARAMETRO INT // controllo argomenti  
while( argv[1][num] != '\0')  
{ if ( (argv[1][num] < '0') || (argv[1][num] > '9') )  
{ printf("Secondo argomento non intero\n");  
printf("Usage Error: %s DimBuffer\n", argv[0]);  
exit(2);  
}  
num++;  
}  
dimBuffer = atoi(argv[1]);
```

# USO DELLE RISORSE PROCESSI (2)

## ESEMPI

- **Dimensionamento Buffer:** scrivere un programma che usi un buffer per il trasferimento di un file

```
#define dim 100
int nread, fdsorg, dimbuff, sd, ...; char buff[dim]; ...
if((fdsorg=open("test.txt", O_WRONLY|O_CREAT|O_TRUNC, 00640))<0)
{ perror("Error opening file \n"); exit(1); }
while( (nread=read(fd_sorg, buff, dimBuffer))>0) write(sd,buff,nread);
// Invio
```

- **Richiedere un intero all'utente in modo ripetuto fino a EOF** durante l'esecuzione del programma (**prevenire errore e ciclo infinito**)

```
int ok; char ch;
while ((ok = scanf("%d", &ch)) != EOF)
{ if (ok != 1) // errore di formato - puntatore scanf bloccato
  { //CONTROLLO
    do {c=getchar(); printf("%c ", c);}
    while (c!= '\n');
    printf("Inserisci un int), EOF per terminare: ");
    continue;
  }
// opera sull'intero
}
```

# USO DELLE RISORSE PROCESSI (3)

---

## Uso di strutture dinamiche anziché statiche

- Quando lo stack delle chiamate eccede i limiti di memoria predefiniti per lo stack del processo, si verifica un errore di tipo **Stack Overflow**
- Anche le chiamate a funzione utilizzano una risorsa di sistema limitata. La causa più comune di stack overflow è l'eccessiva profondità (chiamate innestate) o ricorsione eccessiva all'interno di un programma

Ad **ESEMPIO**:

- Scrivere un programma che mostri il numero massimo di chiamate innestate consentito

```
void ricorsiva (int i)
{
    printf("Nesting level: %d\n", i);
    ricorsiva(++i);
}

int main(int argc, char* argv[])
{
    printf("Testing nested lvls\n");
    ricorsiva(0);
    printf("Test is over\n");
    exit(0);
}
```